## 实验一、哲学家就餐问题

### 一、实验学时：4

### 二、实验内容和目的：

**实验目的**：

掌握哲学家就餐问题，掌握预防死锁的实现方法；了解Linux系统下进程和线程的实现；掌握通过互斥量、POSIX信号量、XSI信号量集实现多线程/多进程同步控制的方法。

**实验内容：**

熟悉Ubuntu系统环境和命令；熟悉Ubuntu系统下的多线程/多进程编程；在Ubuntu系统下编程实现哲学家就餐问题。实现教材2.5.2节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。连续运行30次以上都未出现死锁现象。

### 三、实验原理：

由Dijkstra提出并解决的哲学家进餐问题(The Dinning Philosophers Problem)是典型的同步问题。该问题是描述有五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。

死锁的定义：多个任务循环等待它方占有的资源而无限期地僵持下去的局面。即：行进中的任务，等待永远不可能发生的事件。死锁是一种CPU的状态。

产生死锁的原因：系统资源不足（有限）；进程推进顺序不合适（操作不当）。

同时具备以下四个条件时，死锁就可能发生。

1、互斥条件：

　　 一个资源，在一个时刻只能由一个进程占有；

2、不可抢占条件：

　　 一个资源被占有，在末释放之前，其它进程不得抢占；

3、占有且申请条件：

　　 进程已占有某种资源，又申请新的资源；

4、循环等待条件：

　　 进程已占有某种资源，又要等待其它进程已拥有的资源。

解决死锁的办法有：预防、避免、检测与恢复三种。

死锁预防的基本思想：要求进程申请资源时必须遵守某种协议，从而打破死锁产生的四个必要条件之一。

1、打破互斥条件：

　　允许进程同时访问某些资源，但有些资源不允许同时被访问；

2、打破不可抢占条件：

　　允许进程强行从占有者处抢占某些资源；

　＊以上两个条件的打破都对系统管理不利，所以不易实行，不可取。

3、打破占有且申请条件：

　　采用资源预先分配策略。

　　这个策略是：只有当进程申请的所有资源都得到满足后，才调度其运行。

　　 缺点：利用率低下。

4、打破循环等待条件：

　　实行资源有序分配策略。

　　这个策略是将资源按类编号，按序分配。进程申请资源必须按号递增提出。若一个号已被占有，它只能申请它以后的号。

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式地执行。而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来调协进程对共享资源的访问的。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即P(信号变量))和发送（即V(信号变量))信息操作。最简单的信号量是只能取0和1的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。这里主要讨论二进制信号量。

由于信号量只能进行两种操作等待和发送信号，即P(sv)和V(sv),他们的行为是这样的：

P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行

V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而挂起，就给它加1.

举个例子，就是两个进程共享信号量sv，一旦其中一个进程执行了P(sv)操作，它将得到信号量，并可以进入临界区，使sv减1。而第二个进程将被阻止进入临界区，因为当它试图执行P(sv)时，sv为0，它会被挂起以等待第一个进程离开临界区域并执行V(sv)释放信号量，这时第二个进程就可以恢复执行。

Linux提供了一组精心设计的信号量接口来对信号进行操作，它们不只是针对二进制信号量，下面将会对这些函数进行介绍，但请注意，这些函数都是用来对成组的信号量值进行操作的。它们声明在头文件sys/sem.h中。

1. semget函数

它的作用是创建一个新信号量或取得一个已有信号量，原型为：

1. **int** semget(key\_t key, **int** num\_sems, **int** sem\_flags);

第一个参数key是整数值（唯一非零），不相关的进程可以通过它访问一个信号量，它代表程序可能要使用的某个资源，程序对所有信号量的访问都是间接的，程序先通过调用semget函数并提供一个键，再由系统生成一个相应的信号标识符（semget函数的返回值），只有semget函数才直接使用信号量键，所有其他的信号量函数使用由semget函数返回的信号量标识符。如果多个程序使用相同的key值，key将负责协调工作。

第二个参数num\_sems指定需要的信号量数目，它的值几乎总是1。

第三个参数sem\_flags是一组标志，当想要当信号量不存在时创建一个新的信号量，可以和值IPC\_CREAT做按位或操作。设置了IPC\_CREAT标志后，即使给出的键是一个已有信号量的键，也不会产生错误。而IPC\_CREAT | IPC\_EXCL则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误。

semget函数成功返回一个相应信号标识符（非零），失败返回-1.

1. semop函数

它的作用是改变信号量的值，原型为：

1. **int** semop(**int** sem\_id, **struct** sembuf \*sem\_opa, **size\_t** num\_sem\_ops);

sem\_id是由semget返回的信号量标识符，sembuf结构的定义如下：

1. **struct** sembuf{
2. **short** sem\_num;//除非使用一组信号量，否则它为0
3. **short** sem\_op;//信号量在一次操作中需要改变的数据，通常是两个数，一个是-1，即P（等待）操作，
4. //一个是+1，即V（发送信号）操作。
5. **short** sem\_flg;//通常为SEM\_UNDO,使操作系统跟踪信号，
6. //并在进程没有释放该信号量而终止时，操作系统释放信号量
7. };
8. semctl函数

该函数用来直接控制信号量信息，它的原型为：

1. **int** semctl(**int** sem\_id, **int** sem\_num, **int** command, ...);

如果有第四个参数，它通常是一个union semum结构，定义如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. **union** semun{
2. **int** val;
3. **struct** semid\_ds \*buf;
4. unsigned **short** \*arry;
5. };

前两个参数与前面一个函数中的一样，command通常是下面两个值中的其中一个：

SETVAL：用来把信号量初始化为一个已知的值。p 这个值通过union semun中的val成员设置，其作用是在信号量第一次使用前对它进行设置。

IPC\_RMID：用于删除一个已经无需继续使用的信号量标识符。

1. 进程使用信号量通信

在Linux系统中，使用信号量通常分为以下4个步骤：

   ①  创建信号量或获得在系统中已存在的信号量，此时需要调用 semget() 函数。不同进程通过使用同一个信号量键值来获得同一个信号量。

   ②  初始化信号量，此时使用 semctl() 函数的SETVAL操作。当使用二维信号量时，通常将信号量初始化为1。

   ③  进行信号量的PV操作，此时，调用 semop()函数。这一步是实现进程间的同步和互斥的核心工作部分。

   ④  如果不需要信号量，则从系统中删除它，此时使用semctl()函数的 IPC\_RMID操作。需要注意的是，在程序中不应该出现对已经被删除的信号量的操作。

### 四、实验器材（设备、元器件）

1. 学生每人一台PC，安装WindowsXP/2000操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

### 五、实验步骤：

1. 使用“Ctrl+Alt+T”打开终端；

2. 使用gedit或vim命令打开文本编辑器进行编码： “gedit 文件名.c”

3．编译程序：

“ gcc 文件名.c -o 可执行程序名 ”

（如果只输入 gcc 文件名.c，默认可执行程序名为a.out）

使用线程库时，gcc编译需要添加-lpthread

4. 执行程序：./可执行程序名

## 实验二、生产者/消费者问题的实现

### 一、实验学时：4

### 二、实验内容和目的：

**实验目的**：

掌握进程、线程的概念，熟悉相关的控制语；掌握进程、线程间的同步原理和方法；掌握进程、线程间的互斥原理和方法。

**实验内容：**

有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池：生产者进程从文件中读取一个数据，并将它存放到一个缓冲区中； 消费者进程从一个缓冲区中取走数据，并输出此数据。生产者和消费者之间必须保持同步原则：不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

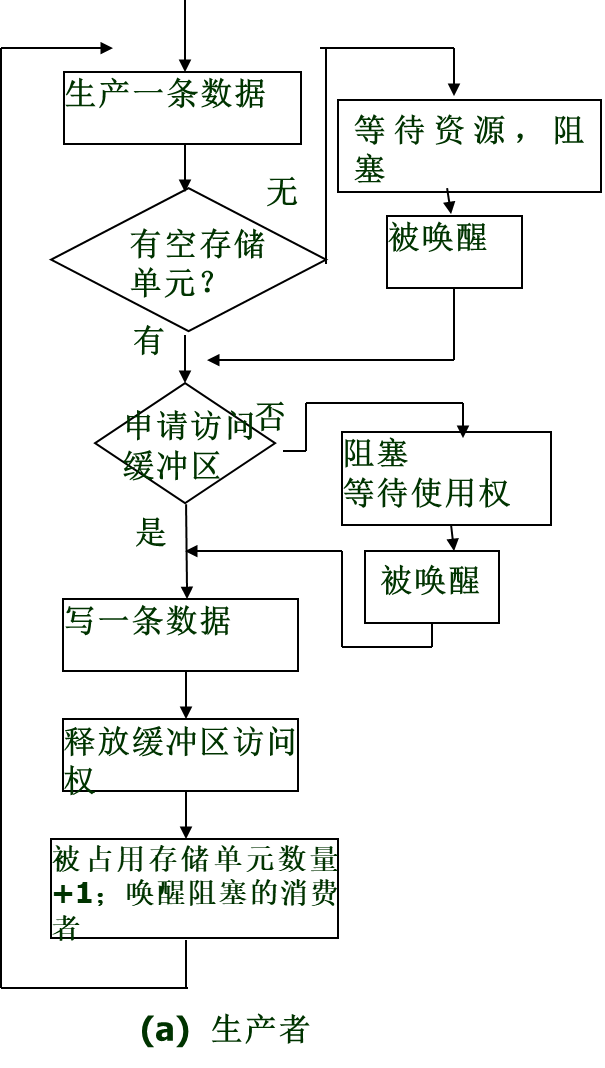
创建3进程（或者线程）作为生产者，4个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为数据。

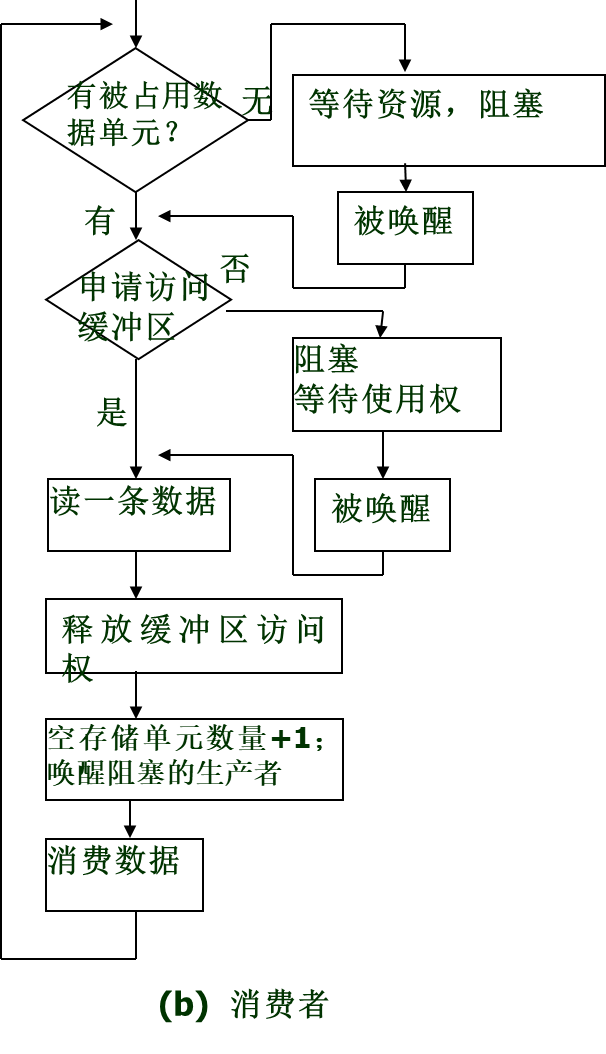
生产者和消费者进程（或者线程）都具有相同的优先级。

### 三、实验原理：

考虑n个缓冲区的缓冲池作为一个共享资源，当生产者进程从数据源—文件中读取数据后将会申请一个缓冲区，并将此数据放入缓冲区中。消费者从一个缓冲区中取走数据，并将其中的内容打印输出。当生产者进程正在访问缓冲区时，消费者进程不能同时访问缓冲区，因此缓冲区是个互斥资源。

生产者、消费者程序执行流程如下图所示：





工程实践证明，利用信号量方法实现进程互斥是高效的，一直被广泛采用，通过信号量机制实现生产者、消费者问题，可以通过以下步骤实现。

步骤1：分配具有n个缓冲区的缓冲池，作为共享资源。

步骤2：定义两个资源型信号量empty 和full，empty信号量表示当前空的缓冲区数量，full表示当前满的缓冲区数量。

步骤3：定义互斥信号量mutex，当某个进程访问缓冲区之前先获取此信号量，在对缓冲区的操作完成后再释放此互斥信号量。以此实现多个进程对共享资源的互斥访问。

步骤4：创建3进程（或者线程）作为生产者，4个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为内容。

步骤5 ：编写代码实现生产者进程的工作内容，即从文件中读取数据，然后申请一个empty信号量，和互斥信号量，然后进入临界区操作将读取的数据放入此缓冲区中。并释放empty信号量和互斥信号量。

步骤6：编写代码实现消费者者进程的工作内容，即先申请一个full信号量，和互斥信号量，然后进入临界区操作从缓冲区中读取数据并打印输出。

Linux系统可以采用的信号量程序接口包括：

1、POSIX命名信号量：使用POSIX IPC名字标识，可用于进程或者线程间的同步。

2、POSIX内存信号量：存放在共享内存中，可用于进程或者线程间的同步。

3、SYSTEM V信号量：在内核中维护，可用于进程或者线程间的同步。

### 四、实验器材（设备、元器件）

1. 学生每人一台PC，安装WindowsXP/2000操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

### 五、实验步骤：

1. 使用“Ctrl+Alt+T”打开终端；

2. 使用gedit或vim命令打开文本编辑器进行编码： “gedit 文件名.c”

3．编译程序：

“ gcc 文件名.c -o 可执行程序名 ”

（如果只输入 gcc 文件名.c，默认可执行程序名为a.out）

使用线程库时，gcc编译需要添加-lpthread

4. 执行程序：./可执行程序名

## 实验三、基于管道的进程间数据传输

### 一、实验学时：4

### 二、实验目的

熟悉Linux下的应用程序开发；熟悉Linux的进程控制原语的使用；掌握Linux操作系统的进程间通信机制管道的使用；掌握Linux操作系统中父进程与子进程的同步。

### 三、实验内容

在Linux系统中使用系统调用fork()创建两个子进程，使用系统调用pipe()建立一个管道，两个子进程分别向管道各写一句话：

Child process 1 is sending a message!

Child process 2 is sending a message!

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。然后分别结束两个子进程的运行。

要求：

1.父进程先接收子进程P1发来的消息，然后再接收子进程P2发来的消息。

2.在Linux平台下实现。

### 四、实验原理

首先创建两个子进程，注意Linux下使用fork()函数创建进程的方法。父进程和两个子进程间需要同步，使用waitpid（）函数实现父进程等待子进程运行完毕后从管道中读取数据并打印，只有子进程将数据写入管道后，父进程才能够执行打开管道操作。

由于fork函数让子进程完整地拷贝了父进程的整个地址空间，所以子进程都有管道的读端和写端。所以在相关进程中最好关掉不用的那一端。根据要求，“父进程先接收子进程P1发来的消息，然后再接收子进程P2发来的消息。”存在两个同步问题，两个子进程和父进程之间（先子写后父读）同步、子进程1和子进程2之间（先1写，再2写）

步骤1：创建一个管道。

步骤2：创建子进程1，向管道中写入“Child process 1 is sending a message!”，并做好跟父进程的同步执行。

步骤3：创建子进程2，向管道中写入“Child process 2 is sending a message!”，并做好跟父进程的同步执行。

步骤4：父进程从管道中读取数据，并打印输出。

### 五、实验步骤

1. 使用“Ctrl+Alt+T”打开终端；

2. 使用gedit或vim命令打开文本编辑器进行编码： “gedit 文件名.c”

3．编译程序：

“ gcc 文件名.c -o 可执行程序名 ”

（如果只输入 gcc 文件名.c，默认可执行程序名为a.out）

使用线程库时，gcc编译需要添加-lpthread

4. 执行程序：./可执行程序名

## 实验四、文件状态测试

### 一、实验学时：4

### 二、实验目的

使学生熟悉UNIX的基本SHELL程序设计方法，包括命令行参数检测、变量设置、文件状态检测与特定信息读取、程序运行控制。

### 三、实验内容

编写一个SHELL程序，动态检测指定文件的状态信息，当文件的大小发生改变时，给出提示信息，并继续前进检测。当文件的大小的变化次数或持续检查无变化次数达到一定值时，退出检查，程序结束。

### 四、实验器材（设备、元器件）

1. 学生每人一台PC，安装WindowsXP/2000操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

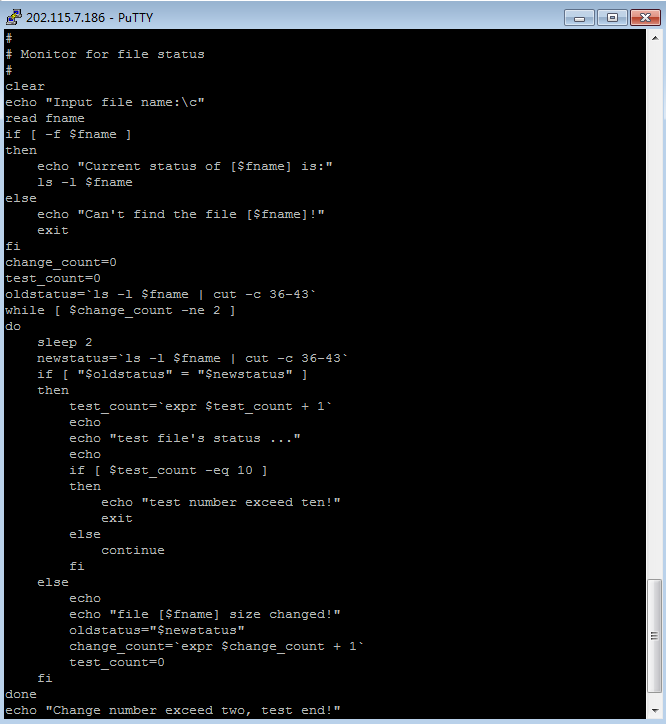
### 五、实验步骤

1. 清屏；
2. 提示用户输入要检测其状态的文件名；
3. 显示该文件的状态信息（提示：该状态信息可由命令ls –l 来得到），或找不到该文件时的错误提示；
4. 可用cut命令，或用sed或awk命令来截取状态信息中文件的大小并保存；
5. 每隔5秒钟检测一次该文件大小的信息，并与保存的文件原来的大小相比较；
6. 如果文件大小未改变，则屏幕显示不变，并继续每隔5秒钟检测一次；
7. 如果文件大小已改变，则保存新的文件大小，并在屏幕上显示：

file [ filename ] size changed

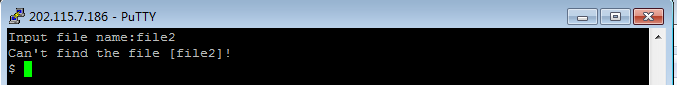
括号中的filename为本程序运行时用户输入的被检测的文件名。程序继续每隔5秒钟检测一次文件的大小；

1. 程序循环执行5~7步的操作。当被检测的文件或者已累计改变了两次大小，或者已连续被检测了十次还未改变大小时，给出相应提示，然后清屏退出。
2. 参考代码如下：

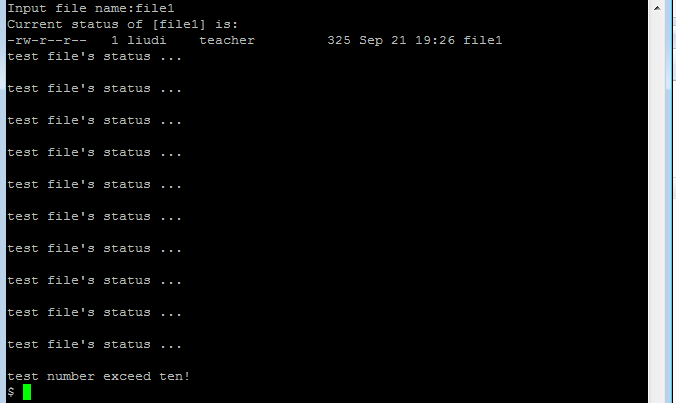


### 六、实验结果

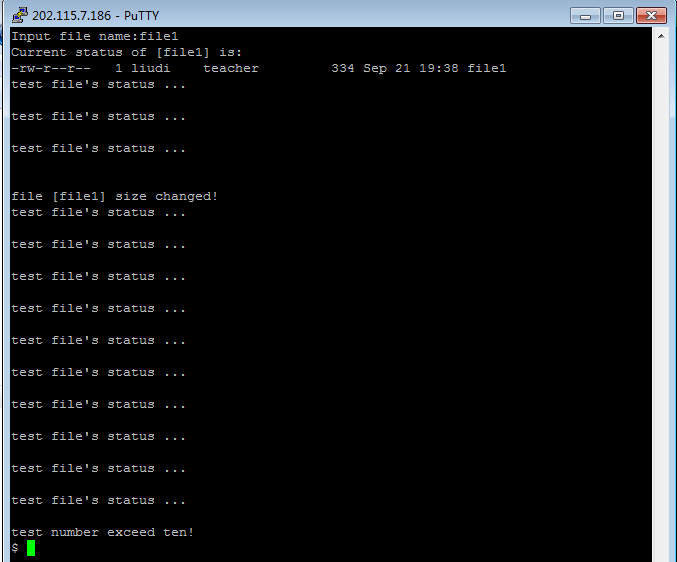
1、程序运行时，如果找不到指定的文件，程序提示错误信息：



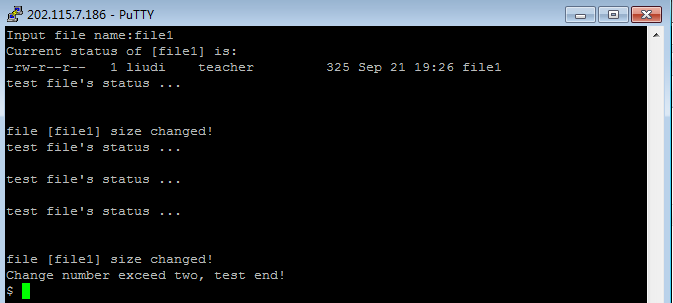
2、运行程序，输入要检查的文件名file1，在十次检查周期内，文件大小都一直不变的情况下，屏幕显示如下：



3、运行程序后，被检查的文件大小只改变了一次时，屏幕的显示情况：



4、运行程序后，被检查的文件的大小改变了两次时，屏幕显示的运行情况：



### 七、实验结论

1、程序运行时，如果找不到指定的文件，程序提示错误信息后退出；

2、程序运行时能够检查指定文件当前的大小值，并每隔5秒钟检查一次；

3、当被检查的文件的大小一直未改变时，连续检查十次后，程序提示相关信息并推出；

4、当被检查的文件大小在检查过程中有修改时，显示相关信息，并重新开始检查计数；

5、当被检查的文件大小发生两次改变时，显示相关信息，并退出程序。

## 实验五、用户动态登录监测

### 一、实验学时：4

### 二、实验目的

使学生熟悉一种简单的指定用户的监测方法. 练习命令行参数、用户变量、while循环控制和暂停进程（sleep）等方面的使用技巧。

### 三、实验内容

编写一个用户监测程序usr\_monitor, 其运行格式为：

usr\_monitor username

其中username是用户指定的任意一个用户名。程序运行时首先列出当前系统中的已登录用户的名单，再检查指定用户是否已登录。如果已登录，则显示相应信息；如果未登录，则等待该用户登录，直到指定用户登录进入系统为止。

### 四、实验器材（设备、元器件）

1. 学生每人一台PC，安装WindowsXP/2000操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

### 五、实验步骤

1. 建立shell程序usr\_monitor
2. 根据变量$#的值检查命令行上是否有一个用户名，如果有则

继续运行；如果没有或多于一个用户名则提示运行格式：

Usage: usr\_monitor username

1. 运行who命令，并将运行结果中的用户名字段截取下来，保存到一个用户变量中。
2. echo命令显示该变量的值，即当前系统中的用户名单。
3. 判断用户名单中是否包含命令行上指定的用户，如果有则显示：

user [username] is logon

并退出程序。

1. 如果用户名单中没有包含命令行上指定的用户，则显示：

waiting user [username] …

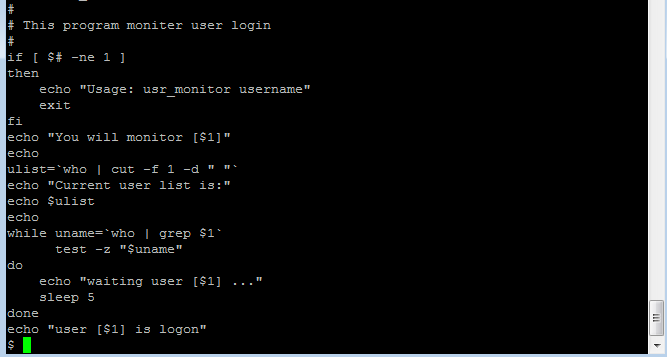
光标停留在省略号后面。

1. 使用while循环，用sleep命令每隔5秒钟检查指定用户是否已登录系统。检测方法与步骤3相同。
2. 如果指定用户未登录，则一直等待下去；如果指定用户已登录，则在省略号后面接着显示：

[username] is log on

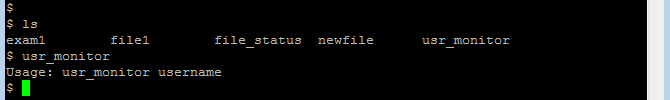
退出循环，并结束程序。

1. 调整程序格式，在程序头加上若干行程序功能注释，在程序的重要功能语句处加上简单注释。
2. 在本实验程序中可以不使用“实验内容”中的命令，只要程序能达到“实验要求”中指定的功能即可。
3. 程序参考代码如下：

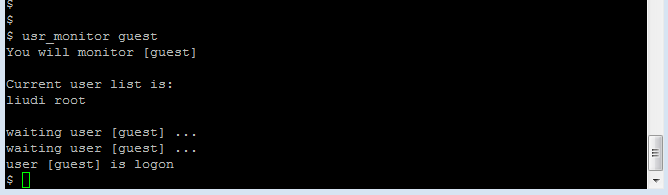


### 六、实验结果

1、命令行上未指定用户名时，程序给出错误提示，并退出：



2、usr\_monitor程序运行时，指定guest用户名，程序显示当前已登录系统的用名单，等待指定的guest用户的登录，当guest登录系统时，程序给出提示信息，并退出。



### 七、实验结论

1、程序usr\_monitor运行时，首先检查命令行上的参数，即是否指定了要被检查的用户名，如果命令行上的参数个数不为一，则提示错误信息后，程序退出；

2、当命令行上指定的有用户名时，程序首先提示要被检查的用户名，并显示当前系统中的活跃用户名单。程序开始等待指定用户的登录，每5秒钟检查一次，一直循环下去，并在屏幕上显示提示信息；

3、当指定的用户登录进入系统后，程序能够检查到相关状态，并在屏幕上显示提示信息，程序退出。

## 实验六、系统命令的实现

### 一、实验学时：4

### 二、实验内容和目的：

**实验目的**：

掌握Linux目录操作方法，包括打开目录、关闭目录、读取目录文件；掌握Linux文件属性获取方法，包括三个获取Linux文件属性的函数、文件属性解析相关的宏；掌握POSIX与ANSI C文件I/O操作方法，包括打开文件、关闭文件、创建文件、读写文件、定位文件。

**实验内容：**

利用POSIX API（文件操作也可以使用ANSI C标准I/O库）编程实现cp –r命令，支持将源路径（目录）中的所有文件和子目录，以及子目录中的所有内容，全部拷贝到目标路径（目录）中。

### 三、实验原理：

1、遍历目录

目录可以被任何有这个目录读权限的人读。然而只有内核才能写一个目录，来保证文件系统的健全。写权限位和执行权限位决定我们是否可以在该目录创建和删除文件－－他们并没有指定我们是否可以写这个目录本身。

目录的真实格式取决于Linux系统实现和文件系统的设计。许多系统避免程序使用read函数来访问目录的内容，因此更加隔离了程序和目录格式的实现相关的细节。

#include <dirent.h>

DIR \*opendir(const char \*pathname);  
成功返回指针，失败返回null。

struct dirent \*readdir(DIR \*dp);  
成功返回指针，目录尾或失败返回null。

void rewinddir(DIR \*dp);  
int closedir(DIR \*dp);

成功返回0，失败返回-1。

long telldir(DIR \*dp);  
返回dp相关的目录的当前位置。

void seekdir(DIR \*dp, long loc);

下面的代码是目录遍历的简单实现：

1. #include <dirent.h>
2. #include <unistd.h>
4. **int**
5. main(**int** argc, **char** \*argv[])
6. {
7. DIR \*dp;
8. **struct** dirent \*dirp;
10. **if** (argc != 2)
11. exit(1);
13. **if** ((dp = opendir(argv[1])) == NULL)
14. exit(1);
15. **while** ((dirp = readdir(dp)) != NULL)
16. printf("%s\n", dirp->d\_name);
17. closedir(dp);
18. exit(0);
19. }

在文件<dirent.h>里定义的dirent结构体依赖于系统实现。实现定义这个结构体至少要包括以下两个成员：  
struct dirent {  
  ino\_t d\_ino;    /\* i-node号 \*/  
  char d\_name[NAME\_MAX + 1]; /\* null结尾的文件名 \*/  
};

DIR结构体是一个内部结构，被那六个函数用来维护关于被读目录的信息。DIR结构体的作用与标准I／O库维护的FILE结构体类似，从opendir返回的DIR结构体的指针被其它5个函数使用。opendir函数初始化所有事情，以便第一个readdir操作读到目录的第一项。在目录里的项的顺序是实现相关的，而且通常不是字母序。

2、Linux文件属性操作

Linux系统的多数文件都是普通文件或者目录，但也有其它的文件类型。文件类型包括：

（1）普通文件。最普遍的文件类型，以某种格式包含数据。对于UNIX内核而言，数据是文本还是二进制没有区别。一个普通文件内容的任何任何解释都留给处理这个文件的程序。一个显著的例外是二进制可执行文件。为了执行一个程序，内核必须了解它的格式。所有的二进制可执行文件都遵守一个允许内核来找到在哪里载入一个程序的代码和数据的格式。

（2）目录文件。一个包含其它文件名和这些文件的信息指针的文件。任何从一个有目录文件读权限的进程都可以读取这个目录的内容，但只有内核才能直接向一个目录文件写入。进程必须使用这章介绍的来对一个目录进行修改。

（3）块特殊文件。一种提供以固定尺寸单位的缓冲I／O方式访问设备（比如硬盘）的文件类型。

（4）字符特殊文件。一种提供以可变尺寸单位的未缓冲I／O方式访问设备的文件类型。在一个系统上的所有设备不是块特殊文件就是字符特殊文件。

（5）FIFO。一种用于进程间通信的文件类型。它有时被称为命令管道。

（6）套接字。一种用于进程间网络通信的文件类型。一个套接字也可以用来在同一主机上的进程间的非网络通信。我们在16章使用套接字来进程进程间的通信。

（7）符号链接。一种指向另一个文件的文件类型。

在Linux中可以通过三个stat函数，来获取文件相关属性信息，包括文件类型、文件权限、文件所属用户、文件最后修改时间等等，如下所示：

int stat(const char \*restrict pathname, struct stat \*restrict buf);

int fstat(int filedes, struct stat \*buf);

int lstat(const char \*restrict pathname, struct stat \*restrict buf);

三个函数成功都返回0，错误都返回-1。

给定一个路径名，stat函数返回一个关于名字对应的文件的信息。fstat函数得到关于已经在描述符filedes上打开的文件的信息。lstat函数与stat相似，但当文件是一个符号链接时，lstat函数返回关于符号链接的信息，而不是其所指向的文件。第二个参数是一个指向一个我们必须提供的结构体的指针。函数填充buf指向的结构体。结构体的定义在不同实现间会有不同，但它看起来大概是：

1. **struct** stat {
2. mode\_t       st\_mode;    /\* 文件类型和模式（权限） \*/
3. ino\_t        st\_ino;     /\* i-node号（序列化号） \*/
4. dev\_t        st\_dev;     /\* 设备号（文件系统） \*/
5. dev\_t        st\_rdev;    /\* 特殊文件的设备号 \*/
6. nlink\_t      st\_nlink;   /\* 链接的数量 \*/
7. uid\_t        st\_uid;     /\* 属主的用户ID \*/
8. git\_t        st\_gid;     /\* 属主的组ID \*/
9. off\_t        st\_size;    /\* 普通文件的以字节为单位的尺寸 \*/
10. **time\_t**       st\_atime;   /\* 最后访问时间 \*/
11. **time\_t**       st\_mtime;   /\* 最后修改时间 \*/
12. **time\_t**       st\_ctime;   /\* 最后文件状态改变时间 \*/
13. blksize\_t    st\_blksize; /\* 最好的I／O块尺寸 \*/
14. blkcnt\_t     st\_blocks;  /\* 分配的磁盘块的数量 \*/
15. };

stat函数用来知道一个文件的所有信息。其中st\_mode的定义如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|  |  |  |  |  | U | G | T | R | W | X | R | W | X | R | W | X |

一个文件的类型编码在stat结构体的st\_mode成员里。我们可以使用下表的宏来决定文件类型。这些宏的参数都是stat结构体的st\_mode成员：

|  |  |
| --- | --- |
| <sys/stat.h>里的文件类型判断宏 | |
| 宏 | 文件类型 |
| S\_ISREG() | 普通文件 |
| S\_ISDIR() | 目录文件 |
| S\_ISCHR() | 字符特殊文件 |
| S\_ISBLK() | 块特殊文件 |
| S\_ISFIFO() | 管道或FIFO |
| S\_ISLNK() | 符号链接 |
| S\_ISSOCK() | 套接字 |

3、Linux文件I/O操作

（1）open函数（open Function）

通过调用open函数可以打开或创建一个文件：

int open(const char \*pathname, int oflga, .../\* mode\_t mode \*/ );

如果成功返回文件描述符，错误返回-1。

pathname是打开或创建的文件的名字。第三个参数为...，是ISO C表示剩余参数的数量和类型可变的方法。对于这个函数来说，第三个参数仅当创建一个文件时使用，我们稍后讨论。我们在函数原型中把这个参数作为一个注释。

这个函数有一个选项群，由参数oflag描述。这个参数由一个或多个以下定义在<fcntl.h>里的常量的通过位或组合而成：

* O\_RDONLY：只读方式打开；
* O\_WRONLY：只写方式打开；
* O\_RDWR：读写方式打开。

为了与早期的程序兼容，多数实现把O\_RDONLY定义为0、O\_WRONLY定义为1、而O\_RDWR定义为2。

这三个常量必须有且只能有一个被指定。以下的常量是可选的：

* O\_APPEND：每次写时添加到文件末尾；
* O\_CREAT：创建这个文件如果它不存在。这个选项需要为open函数提供第三个参数－－打开模式－－来指定这个新建文件的访问权限位
* O\_EXCL：如果O\_CREAT也同时被指定而文件已存在的话会产生一个错误 。这个文件是否存在的测试以及当文件不存在时文件的创建是一个原子操作。
* O\_TRUNC：如果文件存在而且它以只写或读写方式被成功打开的话，把它的长度截为0；
* O\_NOCTTY：如果路径名指向一个终端设备，则不要为这个进程把这个设备作为控制终端分配；
* O\_NONBLOCK：如果文件名指向一个FIFO，一个块特殊文件或一个字符特殊文件，这个选项为打开文件及后续I/O设置非阻塞模式。

（2）creat函数（creat function）

一个新文件也可以通过调用creat函数来创建：

int creat(cont char \*pathname, mode\_t mode);

成功则返回打开的只写文件描述符，错误返回-1，这个函数等同于：

open (pathname, O\_WRONLY | O\_CREAT | O\_TRUNC, mode);

历史上早期的UNIX版本，open的第二个参数只能是0、1或2。当时不可能用open函数来打开一个不存在的文件。因此，一个独立的系统调用－－creat，被用来创建新的文件。现在O\_CREAT和O\_TRUNC选项可以在open里使用，独立的creat已经不再需要了。

当我们在后面的内容中深入讨论文件访问权限时再来说明如何指定mode。

creat的一个缺点是打开的文件只能用来写。在open的新版本出现之前，如果我们想打开一个临时文件写入然后再读回，我们必须调用creat，close，然后再是open。一个更好的方式是使用open函数：

open (pathname, O\_RDWR | O\_CREAT | O\_TRUNC, mode);

（3）close函数（close function）

一个打开的文件可以通过调用close函数来关闭：

int close(int filedes);

成功返回0，错误返回-1。

关闭一个文件会同时释放进程在这个文件上可能会有的任何记录锁（record locks）。当一个进程终止时，它所有的打开文件都被内核自动关闭。许多程序利用这个便利而不显式地关闭打开的文件。

（4）lseek函数

每个打开的文件都有一个相应的“当前文件偏移量（current file offset）”，一般是一个表示从文件开头开始的字节数量的非负整数。读和写的操作一般从当前文件偏移量开始，而且会根据读出或写入的字节数导致偏移量的增加。当文件打开时，这个偏移量默认初始化为0，除非O\_APPEND选项被指定。

一个打开的文件的偏移量可以通过调用lseek显示地设置：

off\_t lseek(int fileds, off\_t offset, int whence);

成功返回新的文件偏移，错误返回-1。

offset的解释取决于whence参数：

如果whence为SEEK\_SET，那文件偏移量会设置为从文件开头之后的offset字节数的位置；如果whence为SEEK\_CUR，那文件偏移量会设置为当前的偏移量加上offset的值。offset可以是正也可以是负的；如果whence为SEEK\_END，那文件偏移量为文件的尺寸加上offset的值。offset可以是正也可以是负的。

因为一个成功的lseek调用会返回新的文件偏移量，我们可以通过从当前位置seek 0字节来决定当前的偏移量：

off\_t currpos;

durrpos = lseek(fd, 0, SEEK\_CUR);

这个技术同样也可以用来确定一个文件是否有seek的能力。如果文件描述符指向一个管道、FIFO或套接字，lseek把errno设置为ESPIPE并返回1。

通常情况下，一个文件的当前偏移量必须是非负整数。尽管如此，有些设备可能允许负的偏移量。但对于普通文件，偏移量必须是非负的。因为负数偏移量是可能的，我们应该小心比较lseek的返回值，应测试它是否等于-1，而不能测试它是否小于0。

（5）read函数（read Function）

可以使用read函数从打开的文件里读取数据：

ssize\_t read(int filedes, void \*buf, size\_t nbytes);

返回读取到的字节数，文件结尾则返回0，错误返回-1。以下几种情况会造成实际读取的字节数比请求的数量少：

在读一个普通文件时，如果在读入请求的字节数之前碰到了文件尾。比如，在请求读取100字节时，只剩30字节就到文件末尾了，read返回30。下次我们调用read时，它会返回0（文件末尾。）

当从一个终端设备读数据时。通常，一次最多只能读一行。（我们将在18章展示如何改变它。）

当从网络上读数据时。网络的缓冲区可能会导致读入的比请求的少。

当从一个管道或FIFO中读数据时。如果管道包含比请求量少的字节，read会只返回可用的字节数。

当从一个面向记录（record-oriented）设备读入数据时。一些面向记录的设备，比如磁带（magnetic type），一次可以最多返回一个记录。

当被一个信号中断，而只读了部分数据时

read操作从文件当前偏移量开始。在成功返回前，偏移量会增加实际读取的字节数。

（6）write函数（write Function）

可能用write函数向一个打开的文件里写数据：

#include <unistd.h>

ssize\_t write(int filedes, const void \*buf, size\_t nbytes);

成功返回写入的字节数，错误返回-1。返回值一般与nbytes参数相等，不然就说明出错了。write错误的一个常见的原因可能是磁盘写满或超过了一个进程的文件大小限制。

对于一个普通文件，写操作从文件当前偏移量开始。如果当文件被打开时O\_APPEND选项被指定，每次写操作前，文件的当前偏移量会设在在文件末尾。在成功写入后，文件偏移量会增加真正写入的字节数。

### 四、实验器材（设备、元器件）

1. 学生每人一台PC，安装WindowsXP/2000操作系统。
2. 个人PC安装VMware虚拟机和Ubuntu系统。

### 五、实验步骤：

1. 使用“Ctrl+Alt+T”打开终端；

2. 使用gedit或vim命令打开文本编辑器进行编码： “gedit 文件名.c”

3．编译程序：

“ gcc 文件名.c -o 可执行程序名 ”

（如果只输入 gcc 文件名.c，默认可执行程序名为a.out）

使用线程库时，gcc编译需要添加-lpthread

4. 执行程序：./可执行程序名